Primitives for Contract-based Synchronization

Massimo Bartoletti
Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari
Roberto Zunino
Dipartimento di Ingegneria e Scienza dell'Informazione, Università degli Studi di Trento

We investigate how contracts can be used to regulate the interaction between processes. To do that, we study a variant of the concurrent constraints calculus presented in [2] , featuring primitives for multi-party synchronization via contracts. We proceed in two directions. First, we exploit our primitives to model some contract-based interactions. Then, we discuss how several models for concurrency can be expressed through our primitives. In particular, we encode the π-calculus and graph rewriting.

1 Introduction

A contract is a binding agreement stipulated between two or more parties, which dictates their rights and their duties, and the penalties each party has to pay in case the contract is not honoured.

In the current practice of information technology, contracts are not that different from those legal agreements traditionally enforced in courts of law. Both software and services commit themselves to respect some (typically weak, if not "without any expressed or implied warranty") service level agreement. In the case this is not honoured, the only thing the user can do is to take legal steps against the software vendor or service provider. Since legal disputes may require a lot of time, as well as relevant expenses, such kinds of contracts serve more as an instrument to discourage users, rather than making easier for users to demand their rights.

Recent research has then addressed the problem of devising new kinds of contracts, to be exploited for specifying and automatically regulating the interaction among users and service providers. See e.g. [6, 8, 11, 13, 20], to cite a few. A contract subordinates the behaviour promised by a client (e.g. "I will pay for a service X") to the behaviour promised by a service (e.g. "I will provide you with a service Y"), and *vice versa*. The crucial problems are then how to formalise the concept of contract, how to understand when a set of contracts gives rise to an agreement among the stipulating parties, and how to actually enforce this agreement in an open, and possibly unreliable, environment.

In the Concurrent Constraint Programming (CCP) paradigm [23, 24], concurrent processes communicate through a global constraint store. A process can add a constraint c to the store through the $\mathsf{tell}\,c$ primitive. Dually, the primitive $\mathsf{ask}\,c$ makes a process block until the constraint c is entailed by the store. Very roughly, such primitives may be used to model two basic operations on contracts: a $\mathsf{tell}\,c$ is for publishing the contract c, and an $\mathsf{ask}\,c'$ is for waiting until one has to fulfill some duty c'.

While this may suggest CCP as a good candidate for modelling contract-based interactions, some important features seem to be missing. Consider e.g. a set of parties, each offering her own contract. When some of the contracts at hand give rise to an agreement, all the involved parties accept the contract, and start interacting to accomplish it. A third party (possibly, an "electronic" court of law) may later on join these parties, so to provide the stipulated remedies in the case an infringement to the contract is found. To model this typical contract-based dynamics, we need the ability of making *all* the parties

Bliudze, Bruni, Grohmann, Silva (Eds.):
Third Interaction and Concurrency Experience (ICE 2010)
EPTCS 38, 2010, pp. 67–82, doi:10.4204/EPTCS.38.8

involved in a contract synchronise when an agreement is found, establishing a session. Also, we need to allow an external party to join a running session, according to some condition on the status of the contract.

In this paper we study a variant of [2], an extension of CCP which allows for modelling such kinds of scenarios. Our calculus features two primitives, called fuse and join: the first fuses all the processes agreeing on a given contract, while the second joins a process with those already participating to a contract. Technically, the prefix $\mathsf{fuse}_x\, c$ probes the constraint store to find whether it entails c; when this happens, the variable x is bound to a fresh session identifier, shared among the parties involved in the contract. Such parties are chosen according to a *local minimal fusion* policy. The prefix $\mathsf{join}_x\, c$ is similar, yet it looks for an already existing session identifier, rather than creating a fresh one. While our calculus is undogmatic about the underlying constraint system, in the contract-based scenarios presented here we commit ourselves to using PCL formulae [2] as constraints.

Contributions. Our contribution consists of the following points. In Sect. 2 we study a calculus for contracting processes. Compared to the calculus in [2], the one in this paper differs in the treatment of the main primitives fuse and join, which have a simplified semantics. Moreover, we also provide here a reduction semantics, and compare it to the labelled one. In Sect. 3 we show our calculus suitable for modelling complex interactions of contracting parties. In Sect. 4 we substantiate a statement made in [2], by showing how to actually encode into our calculus some common concurrency idioms, among which the π-calculus [19] and graph rewriting [21]. In Sect. 5 we discuss further differences between the two calculi, and compare them with other frameworks.

2 A contract calculus

We now define our calculus of contracting processes. The calculus is similar to that in [2], yet it diverges in the treatment of the crucial primitives fuse and join. We will detail the differences between the two versions in Sect. 5. Our calculus features both *names*, ranged over by $n, m, \ldots$, and *variables*, ranged over by $x, y, \ldots$. Constraints are *terms* over variables and names, and include a special element $\bot$; the set of constraints D is ranged over by c, d. Our calculus is parametric with respect to an arbitrary constraint system $(D, \vdash)$ (Def. 1).

Definition 1 (Constraint system [24]) *A constraint system is a pair $(D, \vdash)$, where D is a countable set, and $\vdash \subseteq \mathscr{P}(D) \times D$ is a relation satisfying: (i) $C \vdash c$ whenever $c \in C$; (ii) $C \vdash c$ whenever for all $c' \in C'$ we have $C \vdash c'$, and $C' \vdash c$; (iii) for any c, $C \vdash c$ whenever $C \vdash \bot$.*

Syntax. Names in our calculus behave similarly to the names in the π-calculus: that is, distinct names represent distinct concrete objects. Instead, variables behave as the names in the fusion calculus: that is, distinct variables can be bound to the same concrete object, so they can be fused. A *fusion* σ is a substitution that maps a set of variables to a single name. We write $\sigma = \{^n/_{\vec{x}}\}$ for the fusion that replaces each variable in $\vec{x}$ with the name n. We use metavariables $a, b, \ldots$ to range over both names and variables.

Definition 2 (Processes) *The set of prefixes and processes are defined as follows:*

$$\pi ::= \tau \mid \mathsf{tell}\, c \mid \mathsf{check}\, c \mid \mathsf{ask}\, c \mid \mathsf{join}_x\, c \mid \mathsf{fuse}_x\, c \qquad \textit{(prefixes)}$$

$$P ::= c \mid \textstyle\sum_{i \in I} \pi_i.P_i \mid P|P \mid (a)P \mid X(\vec{a}) \qquad \textit{(processes)}$$

Prefixes π include τ (the silent operation as in CCS), as well as tell, check and ask as in Concurrent Constraints [24]. The prefix $\mathsf{tell}\, c$ augments the context with the constraint c. The prefix $\mathsf{check}\, c$ checks

if c is consistent with the context. The prefix $\mathsf{ask}\,c$ causes a process to stop until the constraint c is entailed by the context. The prefixes $\mathsf{fuse}_x\,c$ and $\mathsf{join}_x\,c$ drive the fusion of the variable x, in two different flavours. The prefix $\mathsf{join}_x\,c$ instantiates x to *any known name*, provided that after the instantiation the constraint c is entailed. The prefix $\mathsf{fuse}_x\,c$ fuses x with *any set of known variables*, provided that, when all the fused variables are instantiated to a fresh name, the constraint c is entailed. To avoid unnecessary fusion, the set of variables is required to be minimal (see Def. 7). To grasp the intuition behind the two kinds of fusions, think of names as session identifiers. Then, a $\mathsf{fuse}_x\,c$ initiates a new session, while a $\mathsf{join}_x\,c$ joins an already initiated session.

Processes P include the constraint c, the summation $\sum_{i\in I}\pi_i.P_i$ of guarded processes over indexing set I, the parallel composition $P|Q$, the scope delimitation $(a)P$, and the instantiated constant $X(\vec{a})$, where $\vec{a}$ is a tuple of names/variables. When a constraint c is at the top-level of a process, we say it is *active*. We use a set of defining equations $\{X_i(\vec{x}) \doteq P_i\}_i$ with the provision that each occurrence of X_j in P_k is guarded, i.e. it is behind some prefix. We shall often use $C = \{c_1, c_2, \ldots\}$ as a process, standing for $c_1|c_2|\cdots$. We write 0 for the empty sum. Singleton sums are simply written $\pi.P$. We use $+$ to merge sums as follows: $\sum_{i\in I}\pi_i.P_i + \sum_{i\in J}\pi_i.P_i = \sum_{i\in I\cup J}\pi_i.P_i$ if $I\cap J=\emptyset$. We stipulate that $+$ binds more tightly than $|$.

Free variables and names of processes are defined as usual: they are free whenever they occur in a process not under a delimitation. Alpha conversion and substitutions are defined accordingly. As a special case, we let $(\mathsf{fuse}_x\,c)\{n/x\} = (\mathsf{join}_x\,c)\{n/x\} = \mathsf{ask}\,c\{n/x\}$. That is, when a variable x is instantiated to a name, the prefixes $\mathsf{fuse}_x\,c$ and $\mathsf{join}_x\,c$ can no longer require the fusion of x, so they behave as a plain $\mathsf{ask}\,c$. Henceforth, we will consider processes up-to alpha-conversion.

We provide our calculus with both a reduction semantics and a labelled transition semantics. As usual for CCP, the former explains how a process evolves within the *whole* context (so, it is not compositional), while the latter also explains how a process interacts with the environment.

Reduction semantics. The structural equivalence relation $\equiv$ is the smallest equivalence between processes satisfying the following axioms:

$$P|0\equiv P \quad P|Q\equiv Q|P \quad P|(Q|R)\equiv(P|Q)|R \quad P+0\equiv P \quad P+Q\equiv Q+P \quad P+(Q+R)\equiv(P+Q)+R$$

$$(a)(P|Q)\equiv P|(a)Q \ \text{ if } a\notin \mathit{free}(P) \qquad (a)(b)P\equiv(b)(a)Q \qquad X(\vec{a})\equiv P\{\vec{a}/\vec{x}\} \ \text{ if } X(\vec{x})\doteq P$$

Definition 3 (Reduction) *Reduction $\to$ is the smallest relation satisfying the rules in Fig. 1.*

We now comment the rules for reduction. Rule TAU simply fires the τ prefix. Rule TELL augments the context (R) with a constraint c. Similarly to [24], we do not check for the consistency of c with the other constraints in R. If desired, a side condition similar to that of rule CHECK (discussed below) can be added, at the cost reduced compositionality. As another option, one might restrict the constraint c in $\mathsf{tell}\,c.P$ to a class of coherent constraints, as done e.g. in [2]. Rule ASK checks whether the context has enough active constraints C so to entail c. Rule CHECK checks the context for consistency with c. Since this requires inspecting every active constraint in the context, a side condition precisely separates the context between C and R, so that all the active constraints are in C, which in this case acts as a global *constraint store*.

Rule FUSE replaces a set of variables $x\vec{y}$ with a bound name n, hence fusing all the variables together. One variable in the set, x, is the one mentioned in the $\mathsf{fuse}_x\,c$ prefix, while the others, $\vec{y}$, are taken from the context. The replacement of variables is done by the substitution σ in the rule premises. The actual set of variables $\vec{y}$ to fuse is chosen according to the *minimal fusion* policy, formally defined below.

Definition 4 (Minimal Fusion) *A fusion $\sigma=\{n/\vec{z}\}$ is* minimal *for C,c, written $C\vdash^{min}_{\sigma} c$, iff:*

$$C\sigma\vdash c\sigma \ \wedge \ \nexists\,\vec{w}\subsetneq\vec{z} \,:\, C\{n/\vec{w}\}\vdash c\{n/\vec{w}\}$$

$$\frac{}{(\vec{a})\,(\tau.P+Q \mid R) \to (\vec{a})\,(P \mid R)}\ [\textsc{Tau}] \qquad \frac{}{(\vec{a})\,(\mathsf{tell}\,c.P+Q \mid R) \to (\vec{a})(c \mid P \mid R)}\ [\textsc{Tell}]$$

$$\frac{C \vdash c}{(\vec{a})\,(C \mid \mathsf{ask}\,c.P+Q \mid R) \to (\vec{a})(C \mid P \mid R)}\ [\textsc{Ask}] \qquad \frac{C, c \not\vdash \bot \quad R \text{ free from active constraints}}{(\vec{a})\,(C \mid \mathsf{check}\,c.P+Q \mid R) \to (\vec{a})\,(P \mid R)}\ [\textsc{Check}]$$

$$\frac{\sigma = \{^{n}/_{xy\vec{y}}\} \quad n \text{ fresh in } P,Q,R,C,c,\vec{a} \quad C \vdash_{\sigma}^{min} c}{(xy\vec{a})\,(C \mid \mathsf{fuse}_x\,c.P+Q \mid R) \to (n\vec{a})\,((C \mid P \mid R)\,\sigma)}\ [\textsc{Fuse}]$$

$$\frac{C\{^{n}/_{x}\} \vdash c\{^{n}/_{x}\}}{(xn\vec{a})\,(C \mid \mathsf{join}_x\,c.P+Q \mid R) \to (n\vec{a})\,\big((C \mid P \mid R)\{^{n}/_{x}\}\big)}\ [\textsc{Join}] \qquad \frac{P \equiv P' \to Q' \equiv Q}{P \to Q}\ [\textsc{Struct}]$$

Figure 1: The reduction relation

A minimal fusion σ must cause the entailment of c by the context C. Furthermore, fusing a proper subset of variables must not cause the entailment. The rationale for minimality is that we want to fuse those variables only, which are actually involved in the entailment of c – not any arbitrary superset. Pragmatically, we will often use $\mathsf{fuse}_x\,c$ as a construct to establish *sessions*: the participants are then chosen among those actually involved in the satisfaction of the constraint c, and each participant "receives" the fresh name n through the application of σ. In this case, n would act as a sort of *session identifier*.

Note that the context R in rule Fuse may contain active constraints. So, the fusion σ is actually required to be minimal with respect to a *subset* C of the active constraints of the whole system. Technically, this will allow us to provide a compositional semantics for $\mathsf{fuse}_x\,c$. Also, this models the fact that processes have a "local" view of the context, as we will discuss later in this section.

Rule Join replaces a variable x with a name n taken from the context. Note that, unlike Fuse, n is *not* fresh here. To enable a $\mathsf{join}_x\,c$ prefix, the substitution must cause c to be entailed by the context C. Intuitively, this prefix allows to "search" in the context for some x satisfying a constraint c. This can also be used to join a session which was previously established by a Fuse.

Note that rules Fuse and Join provide a non-deterministic semantics for prefixes fuse and join since several distinct fusions σ could be used to derive a transition. Each σ involves only names and variables occurring in the process at hand, plus a fresh name n in the case of fuse. If we consider n up-to renaming, we have a finite number of choices for σ. Together with guarded recursion, this makes the transition system to be finitely-branching.

Rule Struct simply allows us to consider processes up-to structural equivalence.

Transition semantics. We now present an alternative semantics, specified through a labelled transition relation. Unlike the reduction semantics, the labelled relation $\xrightarrow{\alpha}$ is *compositional*: all the prefixes can be fired by considering the relevant portion of the system at hand. The only exception is the $\mathsf{check}\,c$ prefix, which is inherently non-compositional. We deal with $\mathsf{check}\,c$ by layering the reduction relation $\rightarrowtail$ over the relation $\xrightarrow{\alpha}$. While defining the transition semantics, we borrow most of the intuitions from the semantics in [2]. The crucial difference between the two is how they finalize the actions generated by a fuse (rule CloseFuse). Roughly, in [2] we need a quite complex treatment, since there we have to accommodate with "principals" mentioned in the constraints. Since here we do not consider principals, we can give a smoother treatment. We discuss in detail such issues in Sect. 5.

We start by introducing in Def. 5 the *actions* of our semantics, that is the set of admissible labels of the LTS. The transition relation is then presented in Def. 6.

Definition 5 (Actions) *Actions α are as follows, where C denotes a set of constraints.*

$$\alpha ::= \tau \mid C \mid C \vdash c \mid C \vdash^F_x c \mid C \vdash^J_x c \mid C \nvdash \bot \mid (a)\,\alpha$$

The action τ represents an internal move. The action C advertises a set of active constraints. The action $C \vdash c$ is a *tentative* action, generated by a process attempting to fire an $\mathsf{ask}\,c$ prefix. This action carries the collection C of the active constraints discovered so far. Similarly for $C \vdash^F_x c$ and $\mathsf{fuse}_x\,c$, for $C \vdash^J_x c$ and $\mathsf{join}_x\,c$, as well as for $C \nvdash \bot$ and $\mathsf{check}\,c$. In the last case, C includes c. The delimitation in $(a)\alpha$ is for scope extrusion, as in the labelled semantics of the π-calculus [22]. We write $(\vec{a})\alpha$ to denote a set of distinct delimitations, neglecting their order, e.g. $(ab) = (ba)$. We simply write $(\vec{a}\vec{b})$ for $(\vec{a} \cup \vec{b})$.

Definition 6 (Transition relation) *The transition relations $\xrightarrow{\alpha}$ are the smallest relations between processes satisfying the rules in Fig. 2. The last two rules in Fig. 2 define the reduction relation $\rightarrowtail$.*

Many rules in Fig. 2 are rather standard, so we comment on the most peculiar ones, only. Note in passing that $\equiv$ is not used in this semantics. The rules for prefixes simply generate the corresponding tentative actions. Rule CONSTR advertises an active constraint, which is then used to augment the tentative actions through the PAR* rules. Rule OPEN lifts a restriction to the label, allowing for scope extrusion. The CLOSE* rules put the restriction back at the process level, and also convert tentative actions into τ.

The overall idea is the following: a tentative action label carries all the proof obligations needed to fire the corresponding prefix. The PAR* rules allow for exploring the context, and augment the label with the observed constraints. The CLOSE* rules check that enough constraints have been collected so that the proof obligations can be discharged, and transform the label into a τ.

The TOP* rules act on the top-level, only, and define the semantics of $\mathsf{check}\,c$.

The side condition of rule CLOSEFUSE involves a variant of the minimal fusion relation we used previously. As for the reduction semantics, we require σ to be minimal, so not to fuse more variables than necessary. Recall however that in the reduction semantics minimality was required with respect to a *part* of the active constraints at hand. In our labelled semantics, rules PAR* collect each active constraint found in the syntax tree of the process. If we simply used $C \vdash^{min}_{\sigma} c$ in CLOSEFUSE, we would handle the following example differently. Let $C = \mathsf{q}(y) \mid \mathsf{q}(z) \vee \mathsf{s} \to \mathsf{p}(y)$, let $P = (x)(y)(z)(\mathsf{fuse}_x\,\mathsf{p}(x).P \mid C \mid \mathsf{s})$, and let $Q = (x)(y)(z)(\mathsf{fuse}_x\,\mathsf{p}(x).P \mid C) \mid \mathsf{s}$. In P we must collect s before applying CLOSEFUSE, and so $\sigma_1 = \{^n/_{xy}\}$ would be the only minimal fusion. Instead in Q we can also apply CLOSEFUSE before discovering s, yielding the minimal fusion $\sigma_2 = \{^n/_{xyz}\}$. This would be inconsistent with $\equiv$ (and our reduction semantics as well). To recover this, we instead require in CLOSEFUSE the following relation, stating that σ must be minimal with respect to a part of the observed constraints, only.

Definition 7 (Local Minimal Fusion) *A fusion $\sigma = \{^n/_{\vec{z}}\}$ is* local minimal *for C, c, written $C \vdash^{loc}_{\sigma} c$, iff:*

$$\exists C' \subseteq C \;:\; C' \vdash^{min}_{\sigma} c$$

While we did not use structural equivalence in the definition of the labelled transition semantics, it turns out to be a bisimulation.

Theorem 1 *The relation $\equiv$ is a bisimulation, i.e.: $P \equiv Q \xrightarrow{\alpha} Q' \implies \exists P'.\, P \xrightarrow{\alpha} P' \equiv Q'$.*

We also have the expected correspondence between the reduction and labelled semantics.

Theorem 2 $P \to P' \iff \exists Q, Q'.\, P \equiv Q \rightarrowtail Q' \equiv P'$

The right implication is by rule induction. To prove the left implication, an induction argument on $Q \xrightarrow{\alpha} Q'$ suffices, exploiting the fact that all the constraints of Q are accumulated in the label.

$$\tau.P \xrightarrow{\tau} P \ \text{[Tau]} \qquad \mathsf{ask}\,c.P \xrightarrow{\emptyset \vdash c} P \ \text{[Ask]} \qquad \mathsf{tell}\,c.P \xrightarrow{\tau} c \,|\, P \ \text{[Tell]}$$

$$\mathsf{check}\,c.P \xrightarrow{\{c\} \nvdash \bot} P \ \text{[Check]} \qquad \mathsf{fuse}_x\,c.P \xrightarrow{\emptyset \vdash^F_x c} P \ \text{[Fuse]} \qquad \mathsf{join}_x\,c.P \xrightarrow{\emptyset \vdash^J_x c} P \ \text{[Join]}$$

$$u \xrightarrow{\{u\}} u \ \text{[Constr]} \qquad \textstyle\sum_i \pi_i.P_i \xrightarrow{\emptyset} \sum_i \pi_i.P_i \ \text{[IdleSum]}$$

$$\frac{P \xrightarrow{(\vec{a})C} P' \quad Q \xrightarrow{(\vec{b})C'} Q'}{P|Q \xrightarrow{(\vec{a}\vec{b})(C \cup C')} P'|Q'} \ \text{[ParConstr]} \qquad \frac{P \xrightarrow{(\vec{a})C} P' \quad Q \xrightarrow{(\vec{b})(C' \vdash c)} Q'}{P|Q \xrightarrow{(\vec{a}\vec{b})(C \cup C' \vdash c)} P'|Q'} \ \text{[ParAsk]}$$

$$\frac{P \xrightarrow{(\vec{a})C} P' \quad Q \xrightarrow{(\vec{b})(C' \vdash^F_x c)} Q'}{P|Q \xrightarrow{(\vec{a}\vec{b})(C \cup C' \vdash^F_x c)} P'|Q'} \ \text{[ParFuse]} \qquad \frac{P \xrightarrow{(\vec{a})C} P' \quad Q \xrightarrow{(\vec{b})(C' \vdash^J_x c)} Q'}{P|Q \xrightarrow{(\vec{a}\vec{b})(C \cup C' \vdash^J_x c)} P'|Q'} \ \text{[ParJoin]}$$

$$\frac{P \xrightarrow{(\vec{a})C} P' \quad Q \xrightarrow{(\vec{b})(C' \nvdash \bot)} Q'}{P|Q \xrightarrow{(\vec{a}\vec{b})(C \cup C' \nvdash \bot)} P'|Q'} \ \text{[ParCheck]} \qquad \frac{P \xrightarrow{\tau} P'}{P|Q' \xrightarrow{\tau} P'|Q'} \ \text{[ParTau]}$$

$$\frac{\pi_j.P_j \xrightarrow{\alpha} P'}{\sum_i \pi_i.P_i \xrightarrow{\alpha} P'} \ \text{[Sum]} \qquad \frac{P\{\vec{a}/\vec{x}\} \xrightarrow{\alpha} P'}{X(\vec{a}) \xrightarrow{\alpha} P'} \ \text{if } X(\vec{x}) \doteq P \ \text{[Def]} \qquad \frac{P \xrightarrow{\alpha} P'}{(a)P \xrightarrow{\alpha} (a)P'} \ \text{if } a \notin \alpha \ \text{[Del]}$$

$$\frac{P \xrightarrow{\alpha} P'}{(a)P \xrightarrow{(a)\alpha} P'} \ \text{[Open]} \qquad \frac{P \xrightarrow{(\vec{a})(C \vdash c)} P'}{P \xrightarrow{\tau} (\vec{a})P'} \ \text{if } C \vdash c \ \text{[CloseAsk]}$$

$$\frac{P \xrightarrow{(x\vec{y}\vec{a})(C \vdash^F_x c)} P' \quad \sigma = \{n/x\vec{y}\} \quad n \text{ fresh} \quad C \vdash^{loc}_{\sigma} c}{P \xrightarrow{\tau} (n\vec{a})(P'\sigma)} \ \text{[CloseFuse]}$$

$$\frac{P \xrightarrow{(xn\vec{a})(C \vdash^J_x c)} P'}{P \xrightarrow{\tau} (n\vec{a})P'\sigma} \ \text{if} \ \begin{array}{l} C\sigma \vdash c\sigma, \\ \sigma = \{n/x\} \end{array} \ \text{[CloseJoin]} \qquad \frac{P \xrightarrow{\tau} P'}{P \rightarrowtail P'} \ \text{[TopTau]}$$

$$\frac{P \xrightarrow{(\vec{a})(C \nvdash \bot)} P'}{P \rightarrowtail (\vec{a})P'} \ \text{if } C \nvdash \bot \ \text{[TopCheck]}$$

Figure 2: The labelled transition relation. Symmetric rules for $+, |$ are omitted. The rules Par* have the following no-capture side condition: $\vec{a}$ is fresh in $\vec{b}, C', c, x, Q'$, while $\vec{b}$ is fresh in C, P'.

3 Examples

We illustrate our calculus by modelling scenarios where the interaction among parties is driven by contracts. In all the examples below, we use as constraints a smooth extension of the propositional contract logic PCL [2]. A comprehensive presentation of PCL is beyond the scope of this paper, so we give here just a broad overview, and we refer the reader to [2, 1] for all the technical details and further examples.

PCL is an extension of intuitionistic propositional logic IPC [25], featuring a *contractual implication* connective $\twoheadrightarrow$. Differently from IPC, a "contract" $\mathsf{b} \twoheadrightarrow \mathsf{a}$ implies a not only when b is true, like IPC implication, but also in the case that a "compatible" contract, e.g. $\mathsf{a} \twoheadrightarrow \mathsf{b}$, holds. So, PCL allows for sort of "circular" assume-guarantee reasoning, summarized by the theorem $\vdash (\mathsf{b} \twoheadrightarrow \mathsf{a}) \wedge (\mathsf{a} \twoheadrightarrow \mathsf{b}) \rightarrow \mathsf{a} \wedge \mathsf{b}$.

The proof system of PCL extends that of IPC with the following axioms:

$$\top \twoheadrightarrow \top \qquad (p \twoheadrightarrow p) \to p \qquad (p' \to p) \to (p \twoheadrightarrow q) \to (q \to q') \to (p' \twoheadrightarrow q')$$

A main result about PCL is its decidability, proved via cut elimination. Therefore, we can use the (decidable) provability relation of PCL as the entailment relation $\vdash$ of the constraint structure.

Example 1 (Greedy handshaking) *Suppose there are three kids who want to play together. Alice has a toy airplane, Bob has a bike, while Carl has a toy car. Each of the kids is willing to share his toy, but only provided that the other two kids promise they will lend their toys to him. So, before sharing their toys, the three kids stipulate a "gentlemen's agreement", modelled by the following* PCL *contracts:*

$$c_{Alice}(x) = (\mathsf{b}(x) \wedge \mathsf{c}(x)) \twoheadrightarrow \mathsf{a}(x) \qquad c_{Bob}(y) = (\mathsf{a}(y) \wedge \mathsf{c}(y)) \twoheadrightarrow \mathsf{b}(y) \qquad c_{Carl}(z) = (\mathsf{a}(z) \wedge \mathsf{b}(z)) \twoheadrightarrow \mathsf{c}(z)$$

Alice's contract $c_{Alice}(x)$ says that Alice promises to share her airplane in a session x, written $\mathsf{a}(x)$, provided that both Bob and Carl will share their toys in the same session. Bob's and Carl's contracts are dual. The proof system of PCL *allows to deduce that the three kids will indeed share their toys in any session n, i.e. $c_{Alice}(n) \wedge c_{Bob}(n) \wedge c_{Carl}(n) \to \mathsf{a}(n) \wedge \mathsf{b}(n) \wedge \mathsf{c}(n)$ is a theorem of* PCL*. We model the actual behaviour of the three kids through the following processes:*

$$Alice = (x)\ \big(\mathsf{tell}\, c_{Alice}(x).\ \mathsf{fuse}_x\, \mathsf{a}(x).\, lendA\big) \qquad Bob = (y)\ \big(\mathsf{tell}\, c_{Bob}(y).\ \mathsf{fuse}_y\, \mathsf{b}(y).\, lendB\big)$$
$$Carl = (z)\ \big(\mathsf{tell}\, c_{Carl}(z).\ \mathsf{fuse}_z\, \mathsf{c}(z).\, lendC\big)$$

A possible trace of the LTS semantics is the following:

$$\begin{aligned}
&Alice \mid Bob \mid Carl \xrightarrow{\tau} (x)\ \big(c_{Alice}(x) \mid \mathsf{fuse}_x\, \mathsf{a}(x).\, lendA\big) \mid Bob \mid Carl \\
&\xrightarrow{\tau} (x)\ \big(c_{Alice}(x) \mid \mathsf{fuse}_x\, \mathsf{a}(x).\, lendA\big) \mid (y)\ \big(c_{Bob}(y) \mid \mathsf{fuse}_y\, \mathsf{b}(y).\, lendB\big) \mid Carl \\
&\xrightarrow{\tau} (x)\ \big(c_{Alice}(x) \mid \mathsf{fuse}_x\, \mathsf{a}(x).\, lendA\big) \mid (y)\ \big(c_{Bob}(y) \mid \mathsf{fuse}_y\, \mathsf{b}(y).\, lendB\big) \mid (z)\ \big(c_{Carl}(z) \mid \mathsf{fuse}_z\, \mathsf{c}(z).\, lendC\big) \\
&\xrightarrow{\tau} (n)\ \big(c_{Alice}(n) \mid lendA\{^n/_x\} \mid c_{Bob}(n) \mid \mathsf{ask}\, \mathsf{b}(n).\, lendB\{^n/_y\} \mid c_{Carl}(n) \mid \mathsf{ask}\, \mathsf{c}(n).\, lendC\{^n/_y\}\big) \\
&\xrightarrow{\tau} (n)\ \big(c_{Alice}(n) \mid lendA\{^n/_x\} \mid c_{Bob}(n) \mid lendB\{^n/_y\} \mid c_{Carl}(n) \mid \mathsf{ask}\, \mathsf{c}(n).\, lendC\{^n/_y\}\big) \\
&\xrightarrow{\tau} (n)\ \big(c_{Alice}(n) \mid lendA\{^n/_x\} \mid c_{Bob}(n) \mid lendB\{^n/_y\} \mid c_{Carl}(n) \mid lendC\{^n/_y\}\big)
\end{aligned}$$

In step one, we use TELL,PARTAU,DEL *to fire the prefix $\mathsf{tell}\, c_{Alice}(x)$. Similarly, in steps two and three, we fire the prefixes $\mathsf{tell}\, c_{Bob}(y)$ and $\mathsf{tell}\, c_{Carl}(z)$. Step four is the crucial one. There, the prefix $\mathsf{fuse}_x\, \mathsf{a}(x)$ is fired through rule* FUSE*. Through rules* CONSTR,PARFUSE*, we discover the active constraint $c_{Alice}(x)$. We use rule* OPEN *to obtain the action $(x)\{c_{Alice}(x)\} \vdash^F_x \mathsf{a}(x)$ for the Alice part. For the Bob part, we use rule* CONSTR *to discover $c_{Bob}(y)$, which we then merge with the empty set of constraints obtained through rule* IDLESUM*; similarly for Carl. We then apply* OPEN *twice and obtain $(y)\{c_{Bob}(y)\}$ and $(z)\{c_{Carl}(z)\}$. At the top level, we apply* PARFUSE *to deduce $(x,y,z)\{c_{Alice}(x), c_{Bob}(y), c_{Carl}(z)\} \vdash^F_x \mathsf{a}(x)$. Finally, we apply* CLOSEFUSE*, which fuses x, y and z by instantiating them to the fresh name n. It is easy to check that $\{c_{Alice}(x), c_{Bob}(y), c_{Carl}(z)\} \vdash^{loc}_{\sigma} \mathsf{a}(x)$ where $\sigma = \{^n/_{xyz}\}$. Note that all the three kids have to cooperate, in order to proceed. Indeed, fusing e.g. only x and y would not allow to discharge the premise $c(z)$ from the contracts c_{Alice} and c_{Bob}, hence preventing any* fuse *prefix from being fired.*

Example 2 (Insured Sale) *A seller S will ship an order as long as she is either paid upfront, or she receives an insurance from the insurance company I, which she trusts. We model the seller contract as the* PCL *formula $s(x) = \mathsf{order}(x) \wedge (\mathsf{pay}(x) \vee \mathsf{insurance}(x)) \twoheadrightarrow \mathsf{ship}(x)$ where x represents the session where the order is placed. The seller S is a recursive process, allowing multiple orders to be shipped.*

$$S \doteq (x)\, \mathsf{tell}\, s(x).\, \mathsf{fuse}_x\, \mathsf{ship}(x).\, (S \mid doShip(x))$$

The insurer contract $i(x) = \mathsf{premium}(x) \twoheadrightarrow \mathsf{insurance}(x)$ *plainly states that a premium must be paid upfront. The associated insurer process I is modelled as follows:*

$$I \doteq (x)\mathsf{tell}\, i(x).\mathsf{fuse}_x\, \mathsf{insurance}(x).\ \big(I \mid \tau.\mathsf{check}\, \neg\mathsf{pay}(x).(\mathit{refundS}(x) \mid \mathit{debtCollect}(x))\big)$$

When the insurance is paid for, the insurer will wait for some time, modelled by the τ *prefix. After that, he will check whether the buyer has not paid the shipped goods. In that case, the insurer will immediately indemnify the seller, and contact a debt collector to recover the money from the buyer. Note that S and I do not explicitly mention any specific buyer. As the interaction among the parties is loosely specified, many scenarios are possible. For instance, consider the following buyers* B_0, B_1, B_2, B_3*:*

$$\begin{array}{lll} b_0(x) = \mathsf{ship}(x) \twoheadrightarrow \mathsf{order}(x) \wedge \mathsf{pay}(x) & & B_0 = (x)\,\mathsf{tell}\, b_0(x).\mathit{receive}(x) \\ b_1(x) = \mathsf{ship}(x) \twoheadrightarrow \mathsf{order}(x) \wedge \mathsf{premium}(x) & & B_1 = (x)\,\mathsf{tell}\, b_1(x).(\mathit{receive}(x) \mid \tau.\mathsf{tell}\, \mathsf{pay}(x)) \\ b_2(x) = \mathsf{order}(x) \wedge \mathsf{pay}(x) & & B_2 = B_0\{^{b_2}/_{b_0}\} \\ b_3(x) = \mathsf{order}(x) \wedge \mathsf{premium}(x) & & B_3 = B_0\{^{b_3}/_{b_0}\} \end{array}$$

The buyer B_0 *pays upfront. The buyer* B_1 *will pay later, by providing the needed insurance. The "incautious" buyer* B_2 *will pay upfront, without asking any shipping guarantees. The buyer* B_3 *is insured, and will not pay. The insurer will then refund the seller, and start a debt collecting procedure. This is an example where a violated promise can be detected so to trigger a suitable recovery action. The minimality requirement guarantees that the insurer will be involved only when actually needed.*

Example 3 (Automated Judge) *Consider an online market, where buyers and sellers trade items. The contract of a buyer is to pay for an item, provided that the seller promises to send it; dually, the contract of a seller is to send an item, provided that the buyer pays. A buyer first issues her contract, then waits until discovering she has to pay, and eventually proceeds with the process CheckOut. At this point, the buyer may either abort the transaction (process NoPay), or actually pay the item, by issuing the constraint* $\mathsf{paid}(x)$*. After the item has been paid, the buyer may wait for the item to be sent (*$\mathsf{ask}\, \mathsf{sent}(x)$*), or possibly open a dispute with the seller (*$\mathsf{tell}\, \mathsf{dispute}(x)$*). Note that, as in the real world, one can always open a dispute, even when the other party is perfectly right.*

$$\begin{aligned} \mathit{Buyer} &= (x)\ \big(\mathsf{tell}\, \mathsf{send}(x) \twoheadrightarrow \mathsf{pay}(x).\ \mathsf{fuse}_x\, \mathsf{pay}(x).\mathit{CheckOut}\big) \\ \mathit{CheckOut} &= \tau.\mathit{NoPay} + \tau.\,\mathsf{tell}\, \mathsf{paid}(x).(\tau.\,\mathsf{tell}\, \mathsf{dispute}(x) + \mathsf{ask}\, \mathsf{sent}(x)) \end{aligned}$$

The behaviour of the seller is dual: issue the contract, wait until she has to send, and then proceed with Ship. There, either choose not to send, or send the item and then wait for the payment or open a dispute.

$$\begin{aligned} \mathit{Seller} &= (y)\ \big(\mathsf{tell}\, \mathsf{pay}(y) \twoheadrightarrow \mathsf{send}(y).\ \mathsf{fuse}_y\, \mathsf{send}(y).\mathit{Ship}\big) \\ \mathit{Ship} &= \tau.\mathit{NoSend} + \tau.\,\mathsf{tell}\, \mathsf{sent}(y).(\tau.\,\mathsf{tell}\, \mathsf{dispute}(y) + \mathsf{ask}\, \mathsf{paid}(y)) \end{aligned}$$

To automatically resolve disputes, the process Judge may enter a session initiated between a buyer and a seller, provided that a dispute has been opened, and either the obligations pay *or* send *have been inferred. This is done through the* join_z *primitive, which binds the variable z to the name of the session established between buyer and seller. If the obligation* $\mathsf{pay}(z)$ *is found but the item has not been paid (i.e.* $\mathsf{check}\, \neg\mathsf{paid}(z)$ *passes), then the buyer is convicted (by* $\mathit{jailBuyer}(z)$*, not further detailed). Similarly, if the obligation* $\mathsf{send}(z)$ *has not been supported by a corresponding* $\mathsf{sent}(z)$*, the seller is convicted.*

$$\begin{aligned} \mathit{Judge} = (z)\ \big(&\mathsf{join}_z\,(\mathsf{pay}(z) \wedge \mathsf{dispute}(z)).\mathsf{check}\, \neg\mathsf{paid}(z).\mathit{jailBuyer}(z) \mid \\ &\mathsf{join}_z\,(\mathsf{send}(z) \wedge \mathsf{dispute}(z)).\mathsf{check}\, \neg\mathsf{sent}(z).\mathit{jailSeller}(z)\big) \end{aligned}$$

A possible trace of the LTS semantics is the following:

$$\begin{aligned}&\mathit{Buyer}\,|\,\mathit{Seller}\,|\,\mathit{Judge} \xrightarrow{\tau}{}^{*} (n)\ \big(\mathsf{send}(n) \twoheadrightarrow \mathsf{pay}(n)\,|\,\mathsf{paid}(n)\,|\,\mathsf{tell}\,\mathsf{dispute}(n)\,|\,\mathsf{pay}(n) \twoheadrightarrow \mathsf{send}(n)\,|\,\mathit{NoSend}\big)\,|\,\mathit{Judge}\\ &\xrightarrow{\tau}{}^{*} (n)\ \big(\mathsf{send}(n) \twoheadrightarrow \mathsf{pay}(n)\,|\,\mathsf{paid}(n)\,|\,\mathsf{dispute}(n)\,|\,\mathsf{pay}(n) \twoheadrightarrow \mathsf{send}(n)\,|\,\mathit{NoSend}\,|\,\mathsf{check}\,\neg\mathsf{sent}(n).\mathit{jailSeller}(n)\,|\cdots\big)\\ &\xrightarrow{\tau}{}^{*} (n)\ \big(\mathit{jailSeller}(n)\,|\cdots\big)\end{aligned}$$

A more complex version of this example is in [2], also dealing with the identities of the principals performing the relative promises. The simplified variant presented here does not require the more general rule for fuse *found in [2].*

Example 4 (All-you-can-eat) *Consider a restaurant offering an all-you-can-eat buffet. Customers are allowed to have a single trip down the buffet line, where they can pick anything they want. After the meal is over, they are no longer allowed to return to the buffet. In other words, multiple dishes can be consumed, but only in a single step. We model this scenario as follows:*

$$\mathit{Buffet} = (x)\ (\mathsf{pasta}(x) \mid \mathsf{chicken}(x) \mid \mathsf{cheese}(x) \mid \mathsf{fruit}(x) \mid \mathsf{cake}(x))$$

$$\mathit{Bob} = (x)\ \mathsf{fuse}_x\,\mathsf{pasta}(x) \wedge \mathsf{chicken}(x).\mathit{SatiatedB} \quad \mathit{Carl} = (x)\ \mathsf{fuse}_x\,\mathsf{pasta}(x).\mathsf{fuse}_x\,\mathsf{chicken}(x).\mathit{SatiatedC}$$

The Buffet can interact with either Bob or Carl, and make them satiated. Bob eats both pasta and chicken in a single meal, while Carl eats the same dishes but in two different meals, thus violating the Buffet policy, i.e.: Buffet $\mid$ *Carl* $\to^*$ *SatiatedC* $\mid P$*. Indeed, the Buffet should forbid Carl to eat the chicken, i.e. to fire the second* fuse_x*. To enforce the Buffet policy, we first define the auxiliary operator* $\oplus$*. Let* $(p_i)_{i\in I}$ *be* PCL *formulae, let* r *be a fresh prime, o a fresh name, and* $z, (z_i)_{i\in I}$ *fresh variables. Then:*

$$\textstyle\bigoplus_{i\in I} p_i = (o)(z)(z_i)_{i\in I}(\mathsf{r}(o,z) \mid \|_{i\in I}\mathsf{r}(o,z_i) \to p_i)$$

To see how this works, consider the process $\oplus_{i\in I} p_i | Q$ *where* Q *fires a* fuse_x *which demands a subset of the constraints* $(p_i)_{i\in J}$ *with* $J \subseteq I$*. To deduce* p_i *we are forced to fuse* z_i *with* z *(and* x*); otherwise we can not satisfy the premise* $\mathsf{r}(o,z_i)$*. Therefore all the* $(z_i)_{i\in J}$ *are fused, while minimality of fusion ensures that the* $(z_i)_{i\in I\setminus J}$ *are not. After fusion we then reach:*

$$(o)(m)\big((z_i)_{i\in I\setminus J}(\,\|_{i\in I\setminus J}\,\mathsf{r}(o,z_i) \to p_i) | \,\|_{i\in J}(\mathsf{r}(o,m) | \mathsf{r}(o,m) \to p_i)\big)\ \mid\ Q'$$

where m *is a fresh name resulting from the fusion. Note that the* $(p_i)_{i\in I\setminus J}$ *can no longer be deduced through fusion, since the variable* z *was "consumed" by the first fusion. The rough result is that* $\oplus_i p_i$ *allows a subset of the* $(p_i)_{i\in I}$ *to be demanded through fusion, after which the rest is no longer available.*

We can now exploit the $\oplus$ *operator to redefine the Buffet as follows:*

$$\mathit{Buffet}' = (x)(\mathsf{pasta}(x) \oplus \mathsf{chicken}(x) \oplus \mathsf{cheese}(x) \oplus \mathsf{fruit}(x) \oplus \mathsf{cake}(x))$$

The new specification actually enforces the Buffet policy, i.e.: $\mathit{Buffet}' \mid \mathit{Carl} \not\to^* \mathit{SatiatedC} \mid P$*. Note that the operator* $\oplus$ *will be exploited in Sect. 4, when we will encode graph rewriting in our calculus.*

4 Expressive power

We now discuss the expressive power of our synchronization primitives, by showing how to encode some common concurrency idioms into our calculus.

Semaphores. Semaphores admit a simple encoding in our calculus. Below, n is the name associated with the semaphore, while x is a fresh variable. $P(n)$ and $V(n)$ denote the standard semaphore operations, and process Q is their continuation.

$$P(n).Q = (x)\ \mathsf{fuse}_x\,\mathsf{p}(n,x).Q \qquad V(n).Q = (x)\ \mathsf{tell}\,\mathsf{p}(n,x).Q$$

Each $\mathsf{fuse}_x\,\mathsf{p}(n,x)$ instantiates a variable x such that $\mathsf{p}(n,x)$ holds. Of course, the same x cannot be instantiated twice, so it is effectively consumed. New variables are furnished by $V(n)$.

Memory cells. We model below a memory cell.

$$\begin{aligned} &\mathit{New}(n,v).Q = (x)\mathsf{tell}\,\mathsf{c}(n,x) \wedge \mathsf{d}(x,v).Q \\ &\mathit{Get}(n,y).Q = (w)\mathsf{fuse}_w\,\mathsf{c}(n,w).\mathsf{join}_y\,\mathsf{d}(w,y).\mathit{New}(n,y).Q \qquad \mathit{Set}(n,v).Q = (w)\mathsf{fuse}_x\,\mathsf{c}(n,w).\mathit{New}(n,v).Q \end{aligned}$$

The process $\mathit{New}(n,v)$ initializes the cell having name n and initial value v (a name). The process $\mathit{Get}(n,y)$ recovers v by fusing it with y: the procedure is destructive, hence the cell is re-created. The process $\mathit{Set}(n,v)$ destroys the current cell and creates a new one.

Linda. Our calculus can model a tuple space, and implement the insertion and retrieval of tuples as in Linda [14]. For illustration, we only consider p-tagged pairs here.

$$\begin{aligned} &\mathit{Out}(w,y).Q = (x)\mathsf{tell}\,\mathsf{p}(x) \wedge \mathsf{p}_1(x,w) \wedge \mathsf{p}_2(x,y).Q \qquad \mathit{In}(w,y).Q = (x)\mathsf{fuse}_x\,\mathsf{p}_1(x,w) \wedge \mathsf{p}_2(x,y).Q \\ &\mathit{In}(?w,y).Q = (x)\mathsf{fuse}_x\,\mathsf{p}_2(x,y).\mathsf{join}_w\,\mathsf{p}_1(x,w).Q \qquad \mathit{In}(w,?y).Q = (x)\mathsf{fuse}_x\,\mathsf{p}_1(x,w).\mathsf{join}_y\,\mathsf{p}_2(x,y).Q \\ &\mathit{In}(?w,?y).Q = (x)\mathsf{fuse}_x\,\mathsf{p}(x).\mathsf{join}_w\,\mathsf{p}_1(x,w).\mathsf{join}_y\,\mathsf{p}_2(x,y).Q \end{aligned}$$

The operation *Out* inserts a new pair in the tuple space. A fresh variable x is related to the pair components through suitable predicates. The operation *In* retrieves a pair by pattern matching. The pattern $\mathit{In}(w,y)$ mandates an exact match, so we require that both components are as specified. Note that fuse will instantiate the variable x, effectively consuming the tuple. The pattern $\mathit{In}(?w,y)$ requires to match only against the y component. We do exactly that in the fuse prefix. Then, we use join to recover the first component of the pair, and bind it to w. The pattern $\mathit{In}(w,?y)$ is symmetric. The pattern $\mathit{In}(?w,?y)$ matches any pair, so we specify a weak requirement for the fusion. Then we recover the pair components.

Synchronous π-calculus. We encode the synchronous π-calculus [19] into our calculus as follows:

$$\begin{aligned} &[P|Q] = [P] \mid [Q] \qquad [(\nu n)P] = (n)[P] \qquad [X(\vec{a})] = X(\vec{a}) \qquad [X(\vec{y}) \doteq P] = X(\vec{y}) \doteq [P] \\ &[\overline{a}\langle b\rangle.P] = (x)\big(\mathsf{msg}(x,b) \mid \mathsf{fuse}_x\,\mathsf{in}(a,x).\,[P]\big) \qquad (x \text{ fresh}) \\ &[a(z).Q] = (y)\big(\mathsf{in}(a,y) \mid (z)\mathsf{join}_z\,\mathsf{msg}(y,z).\,[Q]\big) \qquad (y \text{ fresh}) \end{aligned}$$

Our encoding preserves parallel composition, and maps name restriction to name delimitation, as one might desire. The output cannot proceed with P until x is fused with some y. Dually, the input cannot proceed until y is instantiated to a name, that is until y is fused with some x – otherwise, there is no way to satisfy $\mathsf{msg}(y,z)$.

The encoding above satisfies the requirements of [15]. It is *compositional*, mapping each π construct in a context of our calculus. Further, the encoding is *name invariant* and preserves *termination*, *divergence* and *success*. Finally it is *operationally corresponding* since, writing $\rightarrow_\pi$ for reduction in π,

$$\begin{aligned} P \rightarrow_\pi^* P' &\implies [P] \rightarrowtail^* \sim [P'] \\ [P] \rightarrowtail^* Q &\implies \exists P'.\ Q \rightarrowtail^* \sim [P'] \wedge P \rightarrow_\pi^* P' \end{aligned}$$

where $\sim$ is $\rightarrowtail$-bisimilarity. For instance, note that:

$$[(\nu m)(n\langle m\rangle.P|n(z).Q)] \;\rightarrowtail^*\; (m)(o)\big(\mathsf{msg}(o,m)|[P]|\mathsf{in}(n,o)|[Q]\{^m/_y\}\big) \sim (m)[P|Q\{^m/_y\}]$$

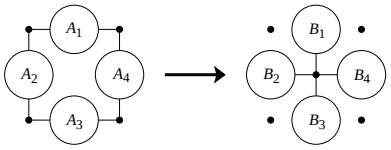
since the name o is fresh and the constraints $\mathsf{msg}(o,m),\mathsf{in}(n,o)$ do not affect the behaviour of P,Q. To see this, consider the inputs and outputs occurring in P,Q. Indeed, in the encoding of inputs, the fuse_x prefix will instantiate x to a fresh name, hence not with o. On the other hand, in the encoding of outputs, the join_z prefix can fire only after y has been fused with x, hence instantiated with a fresh name. The presence of $\mathsf{msg}(o,m)$ has no impact on this firing.

Note that our encoding does not handle non-deterministic choice. This is however manageable through the very same operator $\oplus$ of Ex. 4. We will also exploit $\oplus$ below, to encode graph rewriting.

4.1 Graph rewriting

In the encoding of the π-calculus we have modelled a simple interaction pattern; namely, Milner-style synchronization. Our calculus is also able to model more sophisticated synchronization mechanisms, such as those employed in graph rewriting techniques [21]. Before dealing with the general case, we introduce our encoding through a simple example.

Example 5 *Consider the following "ring-to-star" graph rewriting rule, inspired from an example in [17]:*



Whenever the processes $A_1 \ldots A_4$ are in a configuration matching the left side of the rule (where the bullets represent shared names) a transition is enabled, leading to the right side. The processes change to $B_1 \ldots B_4$, and a fresh name is shared among all of them, while the old names are forgotten. Modelling this kind of synchronization in, e.g., the π-calculus would be cumbersome, since a discovery protocol must be devised to allow processes to realize the transition is enabled. Note that no process is directly connected to the others, so this protocol is non-trivial.

Our calculus allows for an elegant, symmetric translation of the rule above, which is interpreted as an agreement among the processes $A_1 \ldots A_4$. Intuitively, each process A_i promises to change into B_i, and to adjust the names, provided that all the others perform the analogous action. Since each A_i shares two names with the other processes, we write it as $A_i(n,m)$. The advertised contract is specified below as a PCL *formula, where we denote addition and subtraction modulo four as $\boxplus$ and $\boxminus$, respectively:*

$$a_i(n,m,x) = \mathsf{f}_{i\boxplus 1}(x,m) \wedge \mathsf{s}_{i\boxminus 1}(x,n) \twoheadrightarrow \mathsf{f}_i(x,n) \wedge \mathsf{s}_i(x,m) \tag{1}$$

An intuitive interpretation of f,s is as follows: $\mathsf{f}_i(x,n)$ states that n is the first *name of some process $A_i(n,-)$ which is about to apply the rule. Similarly for $\mathsf{s}_i(x,m)$ and the* second *name. The parameter x is a session ID, uniquely identifying the current transition. The contract $a_i(n,m,x)$ states that A_i agrees to fire the rule provided both its neighbours do as well. The actual A_i process is as follows.*

$$A_i(n,m) \doteq (x)\mathsf{tell}\, a_i(n,m,x).\mathsf{fuse}_x\, \mathsf{f}_i(x,n) \wedge \mathsf{s}_i(x,m).B_i(x)$$

Our PCL *logic enables the wanted transition: $P = \|_i A_i(n_i, n_{i\boxplus 1}) \rightarrowtail^* (m)\|_i B_i(m)$.*

Note that the above works even when nodes n_i are shared among multiple parallel copies of the same processes. For instance, $P|P$ will fire the rule twice, possibly mixing A_i components between the two P's.

General Case. We now deal with the general case of a graph rewriting system.

Definition 8 *An hypergraph G is a pair (V_G, E_G) where V_G is a set of vertices and E_G is a set of hyperedges. Each hyperedge $e \in E_G$ has an associated tag $tag(e)$ and an ordered tuple of vertices $(e_1, \ldots, e_k)$ where $e_j \in V_G$. The tag $tag(e)$ uniquely determines the arity k.*

Definition 9 *A graph rewriting system is a set of graph rewriting rules $\{G_i \Rightarrow H_i\}_i$ where G_i, H_i are the* source *and* target *hypergraphs, respectively. No rule is allowed to discard vertices, i.e. $V_{G_i} \subseteq V_{H_i}$. Without loss of generality, we require that the sets of hyperedges E_{G_i} are pairwise disjoint.*

In Def. 10 below, we recall how to apply a rewriting rule $G \Rightarrow H$ to a given graph J. The first step is to identify an embedding σ of G inside J. The embedding σ roughly maps $H \setminus G$ to a "fresh extension" of J (i.e. to the part of the graph that is created by the rewriting). Finally, we replace $\sigma(G)$ with $\sigma(H)$.

Definition 10 *Let $\{G_i \Rightarrow H_i\}_i$ be a graph rewriting system, and let J be a hypergraph. An* embedding σ *of G_i in J is a function such that:* **(1)** *$\sigma(v) \in V_J$ for each $v \in V_{G_i}$, and $\sigma(v) \notin V_J$ for each $v \in V_{H_i} \setminus V_{G_i}$;* **(2)** *$\sigma(e) \in E_J$ for each $e \in E_{G_i}$, and $\sigma(e) \notin E_J$ for each $e \in E_{H_i} \setminus E_{G_i}$;* **(3)** *$\sigma(v) = \sigma(v') \implies v = v'$ for each $v, v' \in V_{H_i} \setminus V_{G_i}$;* **(4)** *$\sigma(e) = \sigma(e') \implies e = e'$ for each $e, e' \in E_{G_i} \cup E_{H_i}$;* **(5)** *$tag(e) = tag(\sigma(e))$ for each $e \in E_{G_i} \cup E_{H_i}$;* **(6)** *$\sigma(e)_h = \sigma(e_h)$ for each $e \in E_{G_i} \cup E_{H_i}$ and $1 \leq h \leq k$.*

The rewriting relation *$J \to K$ holds iff, for some embedding σ, we have $V_K = (V_J \setminus \sigma(V_{G_i})) \cup \sigma(V_{H_i})$ and $E_K = (E_J \setminus \sigma(E_{G_i})) \cup \sigma(E_{H_i})$. The assumption $V_{G_i} \subseteq V_{H_i}$ of Def. 9 ensures $V_J \subseteq V_K$, so no dangling hyperedges are created by rewriting.*

We now proceed to encode graph rewriting in our calculus. To simplify our encoding, we make a mild assumption: we require each G_i to be a *connected* hypergraph. Then, encoding a generic hypergraph J is performed in a compositional way: we assign a unique name n to each vertex in V_J, and then build a parallel composition of processes $A_{tag(e)}(\vec{n})$, one for each hyperedge e in E_J, where $\vec{n} = (n_1, \ldots, n_k)$ identifies the adjacent vertices. Note that since the behaviour of an hyperedge e depends on its tag, only, we index A with $t = tag(e)$. Note that t might be the tag of several hyperedges in each source hypergraph G_i. We stress this point: tag t may occur in distinct source graphs G_i, and each of these may have multiple hyperedges tagged with t. The process A_t must then be able to play the role of any of these hyperedges. The willingness to play the role of such a hyperedge e relatively to a single node n is modelled by a formula $\mathsf{p}_{e,h}(x, n)$ meaning "I agree to play the role of e in session x, and my h-th node is n". The session variable x is exploited to "group" all the constraints related to the same rewriting. We use the formula $\mathsf{p}_{e,h}(x, n)$ in the definition of A_t. The process $A_t(\vec{n})$ promises $\mathsf{p}_{e,1}(x, n_1), \ldots, \mathsf{p}_{e,k}(x, n_k)$ (roughly, "I agree to be rewritten as e"), provided that all the other hyperedges sharing a node n_h agree to be rewritten according to their roles $\bar{e}$. Formally, the contract related to $e \in E_{G_i}$ is the following:

$$a_e(x, \vec{n}) \quad = \bigwedge_{1 \leq h \leq k,\ \bar{e} \in E_{G_i},\ \bar{e_h} = e_h} \mathsf{p}_{\bar{e},\bar{h}}(x, n_h) \twoheadrightarrow \bigwedge_{1 \leq h \leq k} \mathsf{p}_{e,h}(x, n_h) \tag{2}$$

Note that in the previous example we indeed followed this schema of contracts. There, the hypergraph J has four hyperedges e_1, e_2, e_3, e_4, each with a unique tag. The formulae f_i and s_i in (1) are rendered as $\mathsf{p}_{e_i,1}$ and $\mathsf{p}_{e_i,2}$ in (2). Also the operators $\boxplus 1$ and $\boxminus 1$, used in (1) to choose neighbours, are generalized in (2) through the condition $\bar{e_h} = e_h$.

Back to the general case, the process A_t will advertise the contract a_e for each e having tag t, and then will try to fuse variable x. Note that, since the neighbours are advertising the analogous contract, we can not derive any $\mathsf{p}_{e,h}(x, n_h)$ unless *all* the hyperedges in the connected component agree to be rewritten. Since G_i is connected by hypothesis, this means that we indeed require the whole graph to agree.

However, advertising the contracts a_e using a simple parallel composition can lead to unwanted results when non-determinism is involved. Consider two unary hyperedges, which share a node n, and can be rewritten using two distinct rules: $G \Rightarrow H$ with $e1, e2 \in E_G$, and $\overline{G} \Rightarrow \overline{H}$ with $\overline{e1}, \overline{e2} \in E_{\overline{G}}$. Let $tag(e1) = tag(\overline{e1}) = t1$ and $tag(\overline{e2}) = tag(\overline{e2}) = t2$. Each process thus advertises two contracts, e.g.:

$$A_{t1} = (x)\ (a_{e1}(x,n) \mid a_{\overline{e1}}(x,n) \mid Fusion_{t1}) \qquad A_{t2} = (x)\ (a_{e2}(x,n) \mid a_{\overline{e2}}(x,n) \mid Fusion_{t2})$$

Consider now $A_{t1} \mid A_{t2}$. After the fusion of x, it is crucial that both hyperedges agree on the rewriting rule that is being applied – that is either they play the roles of $e1, e2$ or those of $\overline{e1}, \overline{e2}$. However, only one $Fusion$ process above will perform the fusion, say e.g. the first one (the name m below is fresh):

$$(m)(a_{e1}(m,n) | a_{\overline{e1}}(m,n) | Rewrite_{e1} | a_{e2}(m,n) | a_{\overline{e2}}(m,n) | Fusion_{t2}\{^m/_x\})$$

Note that the process $Fusion_{t2}\{^m/_x\}$ can still proceed with the *other* rewriting, since the substitution above cannot disable a prefix which was enabled before. So, we can end up with $Rewrite_{\overline{e2}}$, leading to an inconsistent rewriting. Indeed, A_{t1} was rewritten using $G \Rightarrow H$, while A_{t2} according to $\overline{G} \Rightarrow \overline{H}$.

To avoid this, we resort to the construction $\oplus_i p_i$ discussed in Ex. 4. We can then define A_t as follows.

$$A_t(\vec{n}) \doteq (x)(\bigoplus_{tag(e)=t} a_e(x,\vec{n}) | \sum_{tag(e)=t} \mathsf{fuse}_x \bigwedge_{1 \le h \le k} \mathsf{p}_{e,n}(x,n_h).B_e(x,\vec{n}))$$

In each A_t, the contracts a_e are exposed under the $\oplus$. The consequences of these contracts are then demanded by a sum of fuse_x. We defer the definition of B_e.

Consider now the behaviour of the encoding of a whole hypergraph: $A_t(\vec{n}) | \cdots | A_{t'}(\vec{n}')$. If the hypergraph J contains an occurrence of G, where $G \Rightarrow H$ is a rewriting rule, each of the processes involved in the occurrence $P_1, \ldots, P_l$ may fire a fuse_x prefix. Note that this prefix demands *exactly one* contract a_e from each process inside of the occurrence of G. This is because, by construction, each a_e under the same $\oplus$ involves distinct $\mathsf{p}_{e,n}$. This implies that, whenever a fusion is performed, the contracts which are not directly involved in the handshaking, but are present in the occurrence of G triggering the rewriting, are then effectively disabled. In other words, after a fusion the sums in the other involved processes have exactly one enabled branch, and so they are now committed to apply the rewriting coherently.

After the fusion $B_e(x,\vec{n})$ is reached, where x has been instantiated with a fresh session name m which is common to all the participants to the rewriting. It is then easy to exploit this name m to reconfigure the graph. Each involved vertex (say, with name n) can be exposed to all the participants through e.g. $\mathsf{tell}\,\mathsf{vert}_h(m,n)$, and retrieved through the corresponding $\mathsf{join}_y\,\mathsf{vert}_h(m,y)$. Since m is fresh, there is no risk of interference between parallel rewritings. New vertices (those in $V_H \setminus V_G$) can be spawned and broadcast in a similar fashion. Once all the names are shared, the target hypergraph H is formed by spawning its hyperedges E_H through a simple parallel composition of $A_t(\vec{n})$ processes – each one with the relevant names. Note that the processes A_t, where t ranges over all the tags, are mutually recursive.

Correctness. Whenever we have a rewriting $J \to K$, it is simple to check that the contracts used in the encoding yield an handshaking, so causing the corresponding transitions in our process calculus. The reader might wonder whether the opposite also holds, hence establishing an *operational correspondence*. It turns out that our encoding actually allows *more* rewritings to take place, with respect to Def. 10. Using the A_i from Ex. 5, we have that the following loop of length 8 can perform a transition.

$$P = A_1(n_1,n_2)|A_2(n_2,n_3)|A_3(n_3,n_4)|A_4(n_4,n_5)|A_1(n_5,n_6)|A_2(n_6,n_7)|A_3(n_7,n_8)|A_4(n_8,n_1)$$

Indeed, any edge here has exactly the same "local view" of the graph as the corresponding G of the rewriting rule. So, an handshaking takes place. Roughly, if a graph J_0 triggers a rewriting in the encoding, then each "bisimilar" graph J_1 will trigger the same rewriting.

A possible solution to capture graph rewriting in an exact way would be to mention all the vertices in each contract. That is, edge A_1 would use $p_{A_1}(n_1,n_2,x,y)$, while edge A_2 would use $p_{A_2}(w,n_2,n_3,z)$, and so on, using fresh variables for each locally-unknown node. Then, we would need the fuse prefix to match these variables as well, hence precisely establishing the embedding σ of Def. 10. The semantics of fuse introduced in [2] allows for such treatment.

5 Discussion

We have investigated primitives for contract-based interaction. Such primitives extend those of Concurrent Constraints, by allowing a multi-party mechanism for the fusion of variables which well suites to model contract agreements. We have shown our calculus expressive enough to model a variety of typical contracting scenarios. To do that, we have also exploited our propositional contract logic PCL [2] to deduce the duties inferred from a set of contracts. Finally, we have encoded into our calculus some common idioms for concurrency, among which the π-calculus and graph rewriting.

Compared to the calculus in [2], the current one features a different rule for managing the fusion of variables. In [2], the prefix $\mathsf{fuse}_x\, c$ picks from the context a set of variables $\vec{y}$ (like ours) and a set of *names* $\vec{m}$ (unlike ours). Then, the (minimal) fusion σ causes the variables in $x\vec{y}$ to be replaced with names in $n\vec{m}$, where n is fresh, and $\sigma(x) = n$. The motivation underneath this complex fusion mechanism is that, to establish a session in [2], we need to instantiate the variables $\vec{y}$ which represent the identities of the principals involved in the handshaking. Similarly, $\mathsf{join}_{\vec{x}}\, c$ is allowed to instantiate a set of variables $\vec{x}$. Instead, in this paper, to present our expressivity results we have chosen a simplified version of the calculus, where $\mathsf{fuse}_x\, c$ considers a single name, and $\mathsf{join}_x\, c$ a single variable. At the time of writing, we do not know whether the simplified calculus presented here is as expressive as the calculus of [2]. The contract-related examples shown in this paper did not require the more sophisticated rules for fuse, nor did the encodings of most of the concurrency idioms. As a main exception, we were unable to perfectly encode graph rewriting in our simplified calculus; the difficulty there was that of distinguishing between bisimilar graphs. We conjecture that the more general fusion of [2] is needed to make the encoding perfect; proving this would show our simplified calculus strictly less expressive than the one in [2].

In our model of contracts we have abstracted from most of the implementation issues. For instance, in insecure environments populated by attackers, the operation of exchanging contracts requires particular care. Clearly, integrity of contracts is a main concern, so we expect that suitable mechanisms have to be applied to ensure that contracts are not tampered with. Further, establishing an agreement between participants in a distributed system with unreliable communications appears similar to establishing *common knowledge* among the stipulating parties [16], so an implementation has to cope with the related issues. For instance, the fuse_x prefix requires a fresh name to be delivered among all the contracting parties, so the implementation must ensure everyone agrees on that name. Also, it is important that participants can be coerced to respect their contracts after the stipulation: to this aim, the implementation should at least ensure the non repudiation of contracts [26].

Negotiation and service-level agreement are dealt with in cc-pi [8, 9], a calculus combining features from concurrent constraints and name passing; [7] adds rules for handling transactions. As in the π-calculus, synchronization is channel-based: it only happens between two processes sharing a name. Synchronization fuses two names, similarly to the fusion calculus and ours. A main difference between cc-pi and our calculus is that in cc-pi only two parties may simultaneously reach an agreement, while our fuse allows for simultaneous multi-party agreements. Also, in our calculus the parties involved in an agreement do not have to share a pre-agreed name. This is useful for modelling scenarios where a

contract can be accepted by any party meeting the required terms (see e.g. Ex. 3).

In [12] contracts are CCS-like processes. A client contract is compliant with a service contract if any possible interaction between the client and the service will always succeed, i.e. all the expected synchronizations will take place. This is rather different from what we expect from a calculus for contracts. For instance, consider a simple buyer-seller scenario. In our vision, it is important to provide the buyer with the guarantee that, after the payment has been made, then either the payed goods are made available, or a refund is issued. Also, we want to assure the seller that a buyer will not repudiate a completed transaction. We can model this by the following contracts in PCL: $\mathit{Buyer} = (\mathsf{ship} \vee \mathsf{refund}) \twoheadrightarrow \mathsf{pay}$, and $\mathit{Seller} = \mathsf{pay} \twoheadrightarrow (\mathsf{ship} \vee \mathsf{refund})$. Such contracts lead to an agreement. The contracts of [12] would have a rather different form, e.g. $\mathit{Buyer} = \overline{\mathsf{pay}}.\,(\mathsf{ship} + \mathsf{refund})$ and $\mathit{Seller} = \mathsf{pay}.\,(\overline{\mathsf{ship}} \oplus \overline{\mathsf{refund}})$, where $+$ and $\oplus$ stand respectively for external and internal choice. This models the client outputting a payment, and then either receiving the item or a refund (at service discretion). Dually, the service will first input a payment, and then opt for shipping the item or issuing a refund. This is quite distant from our notion of contracts. Our contracts could be seen as a declarative underspecified description of which behavioural contracts are an implementation. Behavioural contracts seem more rigid than ours, as they precisely fix the order in which the actions must be performed. Even though in some cases this may be desirable, many real-world contracts allow for a more liberal way of constraining the involved parties (e.g., "I will pay before the deadline"). While the crucial notion in [12] is *compatibility* (which results in a yes/no output), we focus on the inferring the *obligations* that arise from a set of contracts. This provides a fine-grained quantification of the reached agreement, e.g. we may identify who is responsible of a contract violation.

Our calculus could be exploited to enhance the compensation mechanism of long-running transactions [4, 5, 10]. There, a transaction is partitioned into a sequence of smaller ones, each one associated with a *compensation*, to be run upon failures of the standard execution. While in long-running transactions clients have little control on the compensations (specified by the designer), in our approach clients can use contracts to select those services offering the desired compensation.

An interesting line for future work is that of comparing the expressiveness of our calculus against other general synchronization models. Our synchronization mechanism, based on the local minimal fusion policy and PCL contracts, seems to share some similarities with the synchronization algebras with mobility [18]. Indeed, in many cases, it seems to be possible to achieve the synchronization defined by a SAM through some handshaking in our model. We expect that a number of SAMs can be encoded through suitable PCL contracts, without changing the entailment relation. Dually, we expect that the interactions deriving from a set of contracts could often be specified through a SAM.

Another general model for synchronization is the BIP model [3]. Here, complex coordination schemes can be defined through an algebra of connectors. While some of these schemes could be modelled by contracts, encoding BIP priorities into our framework seems to be hard. Actually, the only apparent link between priorities and our calculus is the minimality requirement on fusions. However, our mechanism appears to be less general. For instance, BIP allows maximal progress as its priority relation, which contrasts with the minimality of our fusions.

Acknowledgments. Work partially supported by MIUR Project SOFT, and Autonomous Region of Sardinia Project TESLA.

References

[1] Massimo Bartoletti and Roberto Zunino. A logic for contracts. Technical Report DISI-09-034, DISI - Università di Trento, 2009.

[2] Massimo Bartoletti and Roberto Zunino. A calculus of contracting processes. In *Proc. LICS*, 2010.

[3] Simon Bliudze and Joseph Sifakis. The algebra of connectors—structuring interaction in BIP. *IEEE Transactions on Computers*, 57(10), 2008.

[4] Laura Bocchi, Cosimo Laneve, and Gianluigi Zavattaro. A calculus for long running transactions. In *Proc. FMOODS*, 2003.

[5] Roberto Bruni, Hernán C. Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. In *Proc. POPL*, 2005.

[6] Roberto Bruni and Ugo Montanari. Models for open transactions (extended abstract). Technical Report RR-385, Department of Informatics, University of Oslo, 2009.

[7] Maria Grazia Buscemi and Hernán C. Melgratti. Transactional service level agreement. In *Proc. TGC*, 2007.

[8] Maria Grazia Buscemi and Ugo Montanari. CC-Pi: A constraint-based language for specifying service level agreements. In *Proc. ESOP*, 2007.

[9] Maria Grazia Buscemi and Ugo Montanari. Open bisimulation for the Concurrent Constraint Pi-Calculus. In *Proc. ESOP*, 2008.

[10] Michael J. Butler, C. A. R. Hoare, and Carla Ferreira. A trace semantics for long-running transactions. In *Communicating Sequential Processes: The First 25 Years*, 2004.

[11] Samuele Carpineti and Cosimo Laneve. A basic contract language for web services. In *Proc. ESOP*, 2006.

[12] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Trans. on Prog. Lang. and Systems*, 31(5), 2009.

[13] Giuseppe Castagna and Luca Padovani. Contracts for mobile processes. In *Proc. CONCUR*, 2009.

[14] David Gelernter. Generative communication in Linda. *ACM Trans. on Prog. Lang. and Systems*, 7(1), 1985.

[15] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. In *Proc. CONCUR*, 2008.

[16] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3), 1990.

[17] Ivan Lanese and Ugo Montanari. Mapping fusion and synchronized hyperedge replacement into logic programming. *Theory and Practice of Logic Programming*, 7(1-2), 2007.

[18] Ivan Lanese and Emilio Tuosto. Synchronized hyperedge replacement for heterogeneous systems. In *COORDINATION*, 2005.

[19] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1), 1992.

[20] Luca Padovani. Contract-based discovery and adaptation of web services. In *Proc. SFM*, 2009.

[21] Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., 1997.

[22] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[23] Vijay Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[24] Vijay Saraswat, Prakash Panangaden, and Martin Rinard. Semantic foundations of concurrent constraint programming. In *Proc. POPL*, 1991.

[25] Anne Troelstra and Dirk van Dalen. *Constructivism in Mathematics, vol. 1*. North-Holland, 1988.

[26] Jianying Zhou. *Non-repudiation in Electronic Commerce*. Artech House, 2001.